Imperativ Programmering, tentamen

10 januari 2016

- Inga hjälpmedel (böcker eller annat material) är tillåtna.
- Jourhavande lärare: Martin Magnusson. Martin kommer förbi cirka kl 9 för att svara på eventuella frågor.
- Tentan har tre delar, som svarar mot betygskriterierna för betyg 3, 4 och 5.
  - För att bli godkänd, med betyg 3, behövs 20 av 32 poäng från ”3”-frågorna (fråga 1–6) **eller** 32 poäng totalt (från alla frågor).
  - För betyg 4 behövs dessutom 10 av 16 från ”4”-frågorna (fråga 7–10) **eller** 22 poäng tillsammans från fråga 7–12.
  - För betyg 5 behövs dessutom 10 av 16 poäng från ”5”-frågorna (fråga 11–12).

Betyg 3

Fråga 1

8 poäng

Hur många gånger skrivs bokstaven a ut i följande fall (förutsatt att koden exekveras i en `main`-funktion)?

1. ```
   for (int i = 0; i < 10; ++i)
   {
     printf("a");
   }
   ```

2. ```
   for (int i = 1; i < 10; ++i)
   {
     printf("a");
   }
   ```

3. ```
   for (int i = 0; i != 10; i++)
   {
     printf("a");
   }
   ```

4. ```
   for (int i = 0; i < 10; i == i+1 )
   {
     printf("a");
   }
   ```

5. ```
   int i = 0;
   while (i < 10)
   {
     printf("a");
   }
   ```

6. ```
   int i = 0;
   while (i < 10)
   {
     printf("a");
     i++;
   }
   ```

7. ```
   int i = 0;
   while (i = 1)
   {
     printf("a");
     i++;
   }
   ```

8. ```
   int i = 0;
   do
   {
     printf("a");
     i++;
   } while (i == 10);
   ```

Lösning

1. 10

   När loopen börjar är `i` 0, och sista gången villkoret är uppfyllt är när `i` är 9, och då har a skrivits ut 10 gånger.

2. 9

   I den här loopen går `i` från 1 till 9, och alltså skrivs a ut 9 gånger.

3. 10

   Den här loopen stannar när `i` är 10. Eftersom `i` börjar på 0 och räknas upp med 1 i varje varv så blir effekten av att ha `i != 10` densamma som om det hade stått `i < 10`.

4. ∞

   Kruxet med den här loopen är att satsen som görs i slutet av varje varv är en jämförelse och inte en tilldelning. Alltså ändras aldrig `i`, och därför stannar aldrig loopen.

5. ∞

   Även här ändras inte `i`, utan loopen fortsätter så länge `i < 10`, vilket alltid är sant.

6. 10

   Den här loopen gör exakt samma sak som loop nr 1.

7. ∞

   Kruxet med den här loopen är att uttrycket som används som stoppvillkor är en tilldelning, och inte en jämförelse. Uttrycket `i = 1` tilldelar värdet 1 till variabeln `i`, och värdet av uttrycket är också 1, vilket tolkas som sant. Alltså stannar aldrig loopen.

8. 1

   Den här loopen körs så länge `i` är 10, och det är ju aldrig sant i det här fallet. Däremot görs kontrollen i en **`do`**-loop i slutet av varvet, och alltså hinner loopen köra en gång innan den avbryts.

Fråga 2

4 poäng

Fibonacci-talen är en talföljd där varje tal är summan av de båda föregående. Matematiskt kan talföljden definieras som

$$F(n) = \begin{cases} 0 & \text{om } n = 1 \\ 1 & \text{om } n = 2 \\ F(n-1) + F(n-2) & \text{annars.} \end{cases}$$

och de sju första talen är: 0, 1, 1, 2, 3, 5, 8.

(Ett exempel där Fibonaccital dyker upp är i ringarna i fjällen på talkottar.)

Skriv en funktion som tar ett tal n (positivt heltal) som argument, och skriver ut de n första Fibonacci-talen.

Du kan antingen skriva din funktion med C-syntax eller i klartext. Om du väljer att skriva i text är det viktigt att funktionsbeskrivningen ändå är lika detaljerad som den skulle behöva vara i ett C-program. Däremot behöver inte syntaxen vara enligt C. För den här

uppgiften ges inga avdrag för syntaxfel, så länge det är klart vad som är meningen ska hända. Det är inte en övning i C-programmering, men däremot i (imperativt) algoritmiskt tänkande.

(Tips: ett sätt att lösa uppgiften är en loop som itererar n gånger och använder två variabler som håller reda på $F(n-1)$ och $F(n-2)$. Tänk i så fall på att inte råka skriva över någon variabel.)

Lösning

I C kan algoritmen implementeras så här:

```
void fibonacci( int n )
{
  int i = 0;
  int f1, f2; // lagring för f(n-1) respektive f(n-2)
  while (i <= n)
    {
      ++i;
      if (i == 1) // basfall 1
        {
          printf( "%d ", 0 );
          f1 = 0;
          continue;
        }
      if (i == 2) // basfall 2
        {
          printf( "%d ", 1 );
          f2 = f1;
          f1 = 1;
          continue;
        }

      // Standardfallet, när i > 2:
      printf( "%d ", f1+f2 );
      int tmp = f1; // temporär lagring för att inte skriva över f2
      f1 = f2+f1;
      f2 = tmp;
    }
}
```

Nedan följer samma algoritm mer i klartext.

1. Låt $i \leftarrow 1$.
2. Iterera så länge $i \leq n$:
   (a) Om $i = 1$:
      i. Skriv ut 0.
      ii. Gå till rad 2g.
   (b) Om $i = 2$:
      i. Skriv ut 1.
      ii. Låt $f_2 \leftarrow 0$.
      iii. Låt $f_1 \leftarrow 1$.
      iv. Gå till rad 2g.
   (c) Skriv ut $f_1 + f_2$.
   (d) Låt $t \leftarrow f_1$.
   (e) Låt $f_1 \leftarrow f_1 + f_2$.
   (f) Låt $f_2 \leftarrow t$.
   (g) Låt $i \leftarrow i + 1$.

Poängavdrag ges om det inte går att följa algoritmen som angivet och få korrekt resultat, eller om den inte är tillräckligt tydligt eller detaljerat beskrivet.

Jag har gett 0.5 p avdrag om det inte är skrivet som *en* funktion utan två.

Vissa har skrivit en rekursiv funktion som skriver ut värdet av bara det n-te talet, men det är inte rätt. En sådan funktion skulle också kunna anropas n gånger för att skriva ut alla de n första talen. Det är en OK lösning, men den blir ineffektivare än en sekventiell lösning som föreslogs i instruktionerna.

Fråga 3

4 poäng

Datatyper, variabler och tilldelning.

1. *Varför* kan inte talet 314 lagras i en 8-bitars heltalsvariabel?
2. Vilket värde har `x` efter att följande C-kod är exekverad?

```
int a = 1;
int x = (a == 0);
```

3. Vilket värde har `x` efter att följande C-kod är exekverad?

```
float b = 3.1415;
int x = b;
```

4. Vilket värde har `x` efter att följande C-kod är exekverad?

```
int x = 0;
int y = 1;
{
  int x = 10;
  int y = 20;
  x += y;
}
x += y;
```

Lösning

1. Det största tal som går att representera med 8 bitar (binära heltal) är $2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0 = 255$.
2. 0, eftersom uttrycket `a == 0` har värdet 0 (falskt).
3. 3, eftersom `b` typomvandlas till en **`int`** vid tilldelningen, och då avrundas nedåt.
4. 1, eftersom variablerna `x` och `y` som deklareras innanför ”måsvingarna” ligger i ett eget deklarationsområde, och inte påverkar de `x` och `y` som ligger utanför.

Fråga 4

8 poäng

Markera alla fel i följande program. Det finns både syntaxfel (som inte går igenom kompilatorn), och logiska fel (som ger fel resultat). Det är åtta fel totalt.

```
1  /* Ett program som beräknar "x upphöjt till y" för två positiva
2  heltal. */
3
4  #include <stdio.h>
5
6  power( int a; int b )
```

```
7   {
8     float result = 1
9     for {int i = 0; i <= b; ++i}
10      float result;
11      result = result * a;
12
13    return result;
14  }
15
16  int main()
17  {
18    int x = 10;
19    int y = 25;
20
21    // Beräkna x upphöjt till y
22    int z = power( x, y );
23
24    // Skriv ut resultatet
25    printf( "%d ^ %d = %d\n", x, y, result );
26
27    return 0;
28  }
```

Lösning

1. rad 6: Semikolon mellan parametrarna i parameterlistan (syntaxfel).
2. rad 6: Utelämnad returtyp på funktionen. (Är strikt talat ett fel enligt C99, men många kompilatorer nöjer sig med en varning här, eller kanske inte ens det.)
3. rad 8: Inget semikolon som avslutar satsen (syntaxfel).
4. rad 9: {Måsvingar} i stället för (parenteser) i for-satsen (syntaxfel).
5. rad 9: Loopen fortsätter så länge `i <= b` men det borde vara `i < b`, för annars blir det en multiplikation för mycket (logiskt fel). Alternativt kan loopen skrivas `for (int i = 1; i <= b; ++i)`.
6. rad 10: Deklarerar en ny variabel `float result` inuti `for`-loopen, i stället för att använda den tidigare (logiskt fel).
7. rad 11: Den här raden borde ligga i `for`-loopen, men gör inte det eftersom det inte är ett kodblock (sammansatt sats) i loopen. Loopen kommer alltså bara att repetera rad 10, som inte har någon effekt (logiskt fel).
8. rad 19: För stor exponent. Värdet av 10^{25} kommer inte att få plats i en `int` (logiskt fel).
9. rad 25: Variabeln `result` är odefinierad här (syntaxfel).

Här var ett misstag från min sida: det var nio fel totalt, och inte åtta. Men full poäng ges om man har hittat åtta av de nio felen.

Det *borde* också vara med en felkontroll i funktionen `power`, som kollar om `b < 0`, men det blir inte fel i just det här programmet.

Flera har angett att det blir fel att typomvandla `int` till `float` eller vice versa. Det är visserligen osnyggt att göra så, men det är inte fel, och det påverkar inte resultatet i det här programmet.

Fråga 5

4 poäng

Skriv en C-funktion som givet ett nummer (1–7) för en veckodag skriver ut motsvarande

veckodagsnamn (måndag–söndag). Använd ett fält (array) med strängar för att lagra veckodagsnamnen. (Det blir alltså ett fält av **char**-fält.)

Små syntaxfel ger inget poängavdrag i den här uppgiften, men det är viktigt att det framgår hur fält hanteras i C.

Lösning

```
void daystring( int day )
{
  char days[7][8] = {"måndag", "tisdag", "onsdag", "torsdag", "fredag",
                     "lördag", "söndag"};
  printf( "%s", days[day-1]);
}
```

Eftersom strängar i C representeras som fält av **char** måste den här uppgiften implementeras med hjälp av ett tvådimensionellt fält. Det är visserligen möjligt att implementera samma funktionalitet med sju stycken endimensionella fält, men i uppgiften står det ”ett fält”, så jag har gett poängavdrag för en sådan lösning. Jag har också gett poängavdrag för t. ex. `dagar[1]="monday"`, eftersom det inte går att tilldela ett fält till ett annat på det sättet. Det räknar jag inte som ett ”litet syntaxfel”.

Fråga 6

4 poäng

Anta att du får en programbibliotek som innehåller följande funktionsdeklaration.

```
/**
Computes the area of a rectangle.
@param w Width of the rectangle.
@param h Height of the rectangle.
@return If w or h are negative, returns 0. Otherwise returns w*h.
*/
float rect_area( float w, float h );
```

Du har inte tillgång till funktionsdefinitionen, som ligger i en kompilerad objektfil. Ange ett antal testfall (minst 4) för att testa att funktionen verkligen gör det som sägs i kommentaren.

Lösning

Till exempel:

1. `rect_area(10, 10)== 100` (normalfall)
2. `rect_area(5, 2)== 10` (ett annat normalfall)
3. `rect_area(2.5, 2.75)== 2.5*2.75` (ett annat normalfall, med flyttal)
4. `rect_area(10, -10)== 0` (h negativt)
5. `rect_area(-10, 0)== 0` (w negativt)
6. `rect_area(-10, -10)== 0` (båda negativa)

En poäng för varje komplett testfall. För full poäng måste både testfallet och det förväntade resultatet anges. Jag har gett 0.5 p avdrag om bara t. ex. `rect_area(10,10)` har angetts som svar.

Betyg 4

Fråga 7

4 poäng

Vad är en abstrakt datatyp (ADT) och hur kan man implementera en ADT i C?

Lösning

En ADT är en datatyp där den konkreta implementationen har dolts (abstraherats bort) bakom ett *gränssnitt* med funktioner som används för att konstruera och modifiera datatypen, och komma åt dess olika delar.

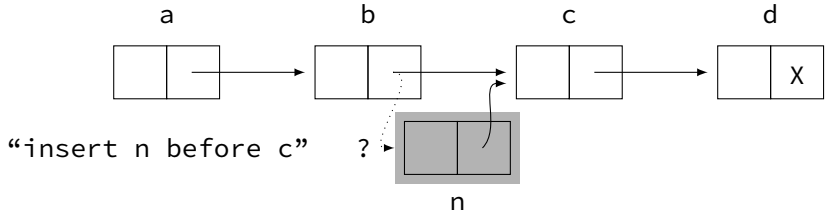
I C implementeras det lämpligtvis genom en *pekare till en struct*, där structen definieras i en separat kompileringsenhet. Användaren behöver då bara hantera en pekare, och de funktioner som hör till gränssnittet för att skapa och modifierar det den pekar på. Den konkreta implementationen (datastrukturen) har abstraherats bort.

Den här frågan var tydligen svår. Många har svarat att en ADT är detsamma som en `typedef`, men det stämmer inte. Det går mycket väl att göra en `typedef int ny_typ ;` eller `typedef struct{int x[10]; int y;} ny_struct;` utan att för den skull abstrahera, eller skilja på, (den konkreta) datatypen och (den abstrakta) beskrivningen av hur datatypen ska användas.

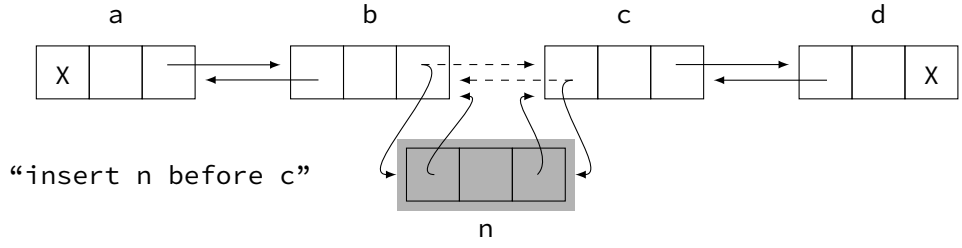
Fråga 8

Ett nackdel med länkade listor är att det inte utan vidare går att lägga till ett element *före* ett annat mitt i listan. Det beror på att varje element bara pekar på nästa. För att lägga in ett nytt element före element nr 3 i en lista med 4 element räcker det då inte att använda en pekare till element nr 3, eftersom också next-pekaren för element nr 2 måste uppdateras för att listan ska hänga ihop.

Bilden nedan visar vad som ska hända. I bilden markerar `X` en null-pekare, och det grå-markerade listelementet är det som ska infogas. Problemet är att det inte går att uppdatera pekaren i det andra elementet (markerad med en prickad pil och ett frågetecken) utan att också gå igenom listan från början för att hitta element nr 2.



Ett sätt att komma runt det är att använda en *dubbellänkad lista*, där varje element har en pekare både till det föregående och efterföljande elementet. Då är det lättare att hitta föregående element för att på så sätt uppdatera pekarna i listan efter att ha lagt till ett element i mitten. Se bilden nedan. Efter infogningen har pekarna uppdaterats. (De streckade pilarna visar på pekarna före infogningen, och de heldragna pilarna efter infogningen.)



2 poäng Skriv en **struct** som är en konkret implementation av en dubbellänkad lista.

3 poäng Skriv en C-funktion som lägger till ett listelement före ett annat, enligt bilden ovan.

Små syntaxfel ger inget poängavdrag i den här uppgiften, men det är viktigt att logiken stämmer, så att det tydligt framgår hur det är tänkt att datastrukturen ska uppdateras.

Lösning

```
1  typedef struct node
2  {
3    struct node *prev;
4    struct node *next;
5    int d;
6  } dlist;
7
8  void insert_before( dlist *i, dlist *new )
9  {
10   new->prev = i->prev;
11   new->next = i;
12   if (i->prev != NULL) // Behövs om i är listans huvud
13   {
14     i->prev->next = new;
15   }
16   i->prev = new;
17 }
```

En nackdel med den här implementationen är att användaren själv måste hålla reda på pekaren till listans huvud. Om `i` är listans huvud innan `insert_before( &i, &n )` så kommer `n` att vara huvudet efteråt. En mer sofistikerad implementation kunde ha en ytterligare struct som håller koll på listans huvud.

För att kunna lägga till ett element först i listan behöver `prev`-pekare kontrolleras (som på rad 12 ovan), men jag har inte gett avdrag för lösningar som missar att kolla det. En snyggare lösning än den ovan borde också kolla om `i` eller `new` är null.

Flera har gjort något i stil med **`int *next;`** i datastrukturen, men det kan inte fungera! För att det ska kunna bli en lista måste varje listelement peka på andra element av samma typ. En **`int`** kan ju inte i sin tur peka vidare på något annat listelement.

Fråga 9

2 poäng Skriv en C-funktion som skriver ut ett fält (array) av heltal. Funktionen ska ta tre argument: ett fält med heltal, ett heltal `n` som anger antalet element, och en öppen textfil (av typen `FILE *`) som heltalen ska skrivas på.

Talen ska skrivas ut med ett mellanslag mellan varje tal. Utmatningen ska avslutas med en punkt och ett radslutstecken (`\n`). Godtyckligt stora fält ska kunna hanteras.

Exempel på utskrifter:

(a) `1 2 9 10 0 33 -6 2 .`

(b) `0 0 0 0 .`

(c) `.`

Här följer deklarationerna av några användbara biblioteksfunktioner.

```
FILE *fopen(const char *path, const char *mode);
int fclose(FILE *stream);
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

2 poäng

Beskriv vad som händer om `n` i själva verket är större respektive mindre än antalet tal som finns i fältet.

Lösning

```
void fun( int arr[], int n, FILE* out )
{
  for (int i = 0; i < n; ++i)
    {
      fprintf( out, "%d ", arr[i] );
    }
  fprintf( out, ".\n" );
}
```

Om `n` är mindre än storleken på `arr` (men större än 0) så kommer bara `n` tecken att skrivas ut. Om `n` är större än storleken på `arr` så är resultatet odefinierat. Antagligen kommer slumpartade tal att skrivas ut när `arr` är slut, och om `n` är väldigt stort kommer programmet att krascha med ett minnesfel.

Fråga 10

3 poäng

För vart och ett av dessa numrerade påståendena om funktioner i C, vilket alternativ är sant? (Bara ett alternativ är sant i varje fall.)

1. En funktion som *anropar sig själv*
   (a) får inte förekomma i C.
   (b) kallas konstruktor.
   (c) kallas iterativ.
   (d) kallas rekursiv.
2. En funktion som *innehåller en annan funktion*
   (a) får inte förekomma i C.
   (b) kallas konstruktor.
   (c) kallas iterativ.
   (d) kallas rekursiv.
3. En funktion i C
   (a) ger alltid ett värde av en bastyp (`int`, `char`, `void`, `struct` etc) som resultat.
   (b) ger alltid ett värde av en bastyp, *eller en pekare till en bastyp* som resultat.
   (c) ger antingen ett värde av en bastyp, eller en pekare till en bastyp, *eller ett fält (array) av bastyper* som resultat.

Lösning

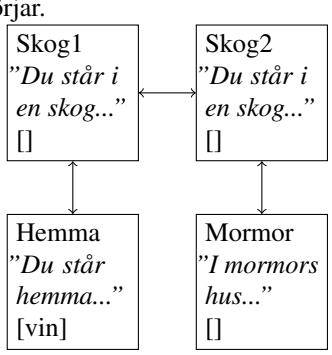
1. ... kallas rekursiv (d).
2. ... får inte förekomma i C (a).
3. ... ger en bastyp eller pekare till bastyp (men inte ett fält) som resultat (b).

Betyg 5

En gång i tiden var textbaserade äventyrsspel populära. Ett typiskt exempel fungerar som följer.

Det finns en karta med platser, där spelaren kan gå mellan olika platser genom att gå åt väster, öster, norr eller söder. Varje plats har en beskrivning (i text) och eventuellt en lista med objekt som kan plockas upp eller användas. Spelaren har också en lista med objekt som hen har plockat upp. Spelet går ut på att gå runt i världen och lösa problem genom att använda olika saker med varandra, för att på så sätt lösa ett uppdrag.

Bilden nedan visar en liten spelvärld som motsvarar sagan om Rödluvan. Spelaren börjar i rutan ”Hemma”, där det ligger en flaska vin. Spelet går ut på att plocka upp vinet, gå genom skogen till mormors hus, och lämna vinet där. I de övriga tre platserna i världen finns inga saker när spelet börjar.



Spelet skulle kunna se ut så här under körning. Användarens inmatning är det som står efter >-tecknet.

```
Hemma.
Du står hemma. Din mamma har bett dig gå med en flaska vin till din
mormor. Ditt hus ligger i en tät skog, men det finns en stig som leder
norrut.
Det finns en flaska vin här.
> ta vin

Hemma.
Du står hemma. Din mamma har bett dig gå med en flaska vin till din
mormor. Ditt hus ligger i en tät skog, men det finns en stig som leder
norrut.
> gå norr

Skog1.
Du står i skogen. En stig leder söderut och österut. Du hör en varg
yla längre in i skogen.
> gå öst

Skog2.
Du står i skogen. En stig leder västerut och söderut. Mot söder ser du
att stigen leder fram till ett hus.
> gå syd

Mormor.
I mormors hus. Din mormor ligger sjuk i sängen.
> släpp vin

Mormor.
I mormors hus. Din mormor ligger sjuk i sängen.
Det finns en flaska vin här.

Du har klarat spelet!
```

Fråga 11

4 poäng

Nu ska du implementera en del av ett sådant här textspel.

Skriv en C-implementation för de datatyper som behövs för världen och spelaren. Du behöver inte tänka på att separera implementation och specifikation här, men definiera hur du i C kan representera en värld enligt exemplet ovan samt en spelperson. Spelet måste alltså kunna veta var spelaren är. Spelet måste också kunna veta hur de olika platserna hänger ihop (åt vilka håll man kan gå från varje plats) och beskrivningen för varje plats. Du behöver inte implementera någon datatyp för att hålla reda på vilka saker som finns på olika ställen, utan bara hur platserna hänger ihop, hur deras beskrivningar lagras, och hur spelet vet var spelaren är.

Ange vilka antaganden du gör om du behöver göra antaganden som inte är specificerade i de här instruktionerna.

2 poäng

Skriv också en funktion (som använder dina datatyper) som flyttar spelaren från en plats till en annan, om det är möjligt. Alltså den funktion som ansvarar för att spelaren flyttas från ”Hemma” till ”Skog1” om hen går norrut från ”Hemma”.

4 poäng

Diskutera också två eller fler alternativa sätt att lagra vilken plats spelaren är på. Hör det till datatypen för spelaren, världen, eller ska det ligga någon annanstans? Vad har det för för- och nackdelar?

4 poäng

Diskutera också två eller fler alternativa sätt att lagra textsträngarna för varje plats. Vad har de för för- och nackdelar?

Lösning

Datatyper Här är ett lösningsförslag där spelaren har en pekare till vilken plats hen befinner sig på.

```
typedef struct place
{
  char name[80];
  char desc[1000];
  //list *items; // hoppa över föremål i denna implementation
  struct place *places[4]; //norr, öst, syd, väst
} place;

typedef struct
{
  place* loc;
  //list *items; // hoppa över föremål i denna implementation
} player;
```

Förflyttningsfunktion Här är ett lösningsförslag som använder implementationen ovan.

```
place* go( place* p, int dir )
{
  if (p->places[dir] != NULL)
    return p->places[dir];
  else return p;
}
```

Den används så här, om spelaren ska (försöka) gå norrut:

```
player p;
place home;
p.loc = &home;

p.loc = go( p.loc, 0 );
```

Spelarens plats En annan variant vore att låta varje plats i världen innehålla en ”flagga” som är 1 om spelaren är där och 0 annars. Det är inte så lyckat, eftersom hela spelvärlden då måste traverseras för att veta var spelaren är.

En ytterligare variant är att ha en fristående `place*`-variabel som inte är en del av spelar-datatypen. Det är i princip likvärdigt med implementationen ovan.

En fjärde variant är att ha en ytterligare **`struct`** `world` som innehåller de olika platserna samt en pekare till vilken plats spelaren är på. Det är funktionellt i princip likvärdigt med implementationen ovan, men möjligtvis lite mer ”prydligt”.

Den största fördelen med implementationen ovan, till skillnad från de övriga som nämns här, är om spelet ska innehålla flera spelare. Då blir det lättare att varje spelare håller reda på var den själv är.

Stränglagring Implementationen ovan använder **`char`**-fält (med bestämd storlek). Fördelen med det är att vi inte behöver hålla reda på `malloc` och `free`, utan att allokering och deallokering sker automatiskt (deallokering när variabelns deklarationsområde tar slut). (Alla strängar ligger på ”stacken” — till skillnad från ”heapen”). Däremot blir det lite krångligt med tilldelning: vi måste använda `strcpy` (eller ännu hellre `strncpy`). Andra nackdelar är att strängarna tar upp onödigt mycket plats (om vi inte behöver 1000 tecken för varje platsbeskrivning, t ex), respektive att de aldrig kan vara längre än den förutbestämda storleken.

En annan variant vore att använda **`char*`** i stället. Då går det lättare att tilldela text till platserna:

```
place start;
start.desc = "Du står hemma...";
```

Här kommer `start.desc` att vara precis lagom stor. Däremot har vi nu gjort så att `start.desc` pekare på en *textsträngsliteral*, vilket betyder att vi inte senare kan ändra strängen hur som helst. Men det är nog inget problem i den här tillämpningen.

En tredje variant vore att allokera “lagom” mycket minne med `malloc` för varje sträng, och sedan göra `strcpy` för att kopiera in strängen dit.

En fjärde variant vore att implementera en bättre sträng-datatyp, som kan växa och krympa efter behov, men det är betydligt krångligare att implementera.

En del har svarat på den här frågan med en diskussion om att lagra strängarna i en separat textfil jämfört med att lagra dem i källkoden. En sådant svar ger också poäng.

Allmänt för de här diskussionsfrågorna gäller att åtminstone två lösningar (varav den ena kan vara den lösning du har använt ovan) ska jämföras, och att det framgår att du förstår hur de fungerar och deras för- och nackdelar.

Fråga 12

2 poäng

För var och en av raderna 1–4 i koden nedan, vilket av alternativen (a)–(f)[1] stämmer?

```
1 int * x1 = malloc( 10 * sizeof( int ) );
2 int * x2 = malloc( 10 );
3 int x3[10];
4 int * x4 = 10;
```

(a) Allokerar plats för 10 **`int`**.

(b) Allokerar plats för 10 bytes.

(c) Allokerar plats för en pekare.

[1] Här blev det lite felskrivet. Det stod (a)–(b) från början, och alternativ (g) var redundant. Det skulle alltså vara (a)–(f).

(d) Allokerar plats för en pekare och 10 `int`.

(e) Allokerar plats för en pekare och 10 bytes.

(f) Allokerar plats för en `int`.

(g) ~~Allokerar plats för 10 `int`.~~

Lösning En halv poäng per rad.

1. (d) en pekare (på stacken) och 10 `int` (på heapen)
2. (e) en pekare (på stacken) och 10 bytes (på heapen)
3. (a)/(g) 10 `int` (på stacken)
4. (c) en pekare (som visserligen också är ett heltal, men som kanske inte använder lika många bytes som en `int`)